A Process-Based Analysis Of Object-Oriented System Analysis And Design

Benjamin Khoo, PhD, New York Institute Of Technology, USA

ABSTRACT

System Analysis and Design (SAND), is critical for any system development project. Most new systems are built using Object-Oriented System Analysis and Design (OOSAND). This paper critically examined and analyzed the OOSAND methodology to discover the underlying principles and rationales based on the inherent processes. There are a few past studies that had examined the factors influencing the processes but few had examined the processes themselves. This paper focuses on the SAND processes and examines the pragmatic issues concerning them. The significance of this research is that the knowledge gained in this exercise will provide systems analyst/programmers a better heuristics to migrate legacy systems to the new object-oriented system and enable higher analyst/programmer efficiency and effectiveness in conducting SAND.

Keywords: Object-Oriented; System Analysis and Design; Process

INTRODUCTION

The advent of the computer has stimulated the development of computer-based systems. As computers became more complex, so are the skills required to develop systems that execute on them. In general, information systems (IS) can be defined as computer-based systems that are required for an organization's business application. The twentieth century saw more complex systems being developed as a result of the advancement in technology. As systems became more complex, more skills were required to develop them. Often IS are developed based on the system development life cycle (SDLC) using elicited user requirements. Today IS development is a specialized skill area with a body of knowledge that is taught in different courses in the curriculum of most universities. The SDLC is made up of the planning, analysis, design, implementation and maintenance phases. While each of the phases needs to be performed in all cases, there are a number of ways their interactions can be organized. The major models are the Waterfall Model (Royce, 1970; Boehm, 1976), Spiral model (Boehm, 1988) and the Unified Process model (Jacobson, 1999; Kruchten, 2000). It is known that 80% of the resources need to be spent on the analysis and design phases, and 20% spent on the implementation phase (Pareto's Rule). The front-end planning, analysis and design phases are critical for any IS development project. The outcome of this is the system design. System design usually means the design of software; sometimes it can also include the design of computer hardware as well. In this paper, system design refers to the design of software.

A methodology can be defined as the underlying principles and rules that govern a system. A process can be defined as a systematic procedure for a set of activities. Thus, from these definitions, a methodology will encompass the processes used within the methodology. Svoboda (1990) developed the idea of a methodology further by proposing that there should be at least four components:

1. a conceptual model of constructs essential to the problem,
2. a set of procedure suggesting the direction and order to proceed,
3. a series of guidelines identifying things to be avoided, and
4. a collection of evaluation criteria for assessing the quality of the product.

The conceptual model is needed to direct or guide the designers to the relevant aspects of the system. The set of procedure provides the designer a systematic and logical set of activities to begin the design task. The evaluation criteria provide an objective measurement of the work done against some established standard or specifications.

SAND methodology provides a logical and systematic means of proceeding with the analysis and design phases as well as a set of guidelines for decision-making. The methodology provides a sequence of activities, and often uses a set of notations or diagrams. The methodology is especially important for large complex projects that involve programming-in-the-large (where many designers are involved); the use of a methodology establishes a set of common communication channels for translating design to codes and a set of common objectives. In addition, there must be an objective match between the overall character of the problem and the features of the solution approach, in order for a methodology to be efficient. There is currently a proliferation of methods in the areas of IS development. Arguably, the two major methodologies for systems design today are the structured systems analysis and design (SSAND), and the object-oriented system analysis and design (OOSAND). The SSAND methodology has not changed much over the years since its inception. As a matter of fact, most legacy systems today were developed using SSAND. There are inherent processes within the various phases in each of these methodologies. This paper will critically examine and analyze the design phase of these two methodologies so as to discover the underlying principles based on their inherent processes.

SYSTEM DESIGN

Many software development projects have been known to incur extensive and costly design errors. The most expansive errors are often introduced early in the development process. This underscores the need for better requirement definition and software design. Software design is an important activity as it determines how the whole software development task would proceed including the system maintenance. The design of software is essentially a skill, but it usually requires a structure that serves as a guide or a methodology for this task.

A system design can be structured as comprising of the software design process component and the software design representation component. The process component is based on the basic principles established in the methodology while the representation component is the "blueprint" from which the code for the software will be built. The evolution of each software design needs to be meticulously recorded or diagramed, including the basis for choices made, for future walk-through and maintenance. It should be noted, that in practice, the design process is often constrained by existing hardware configuration, the implementation language, the existing file and data structures and the existing company practices, all of which would limit the solution space available to develop the software.

Successful design often begins with a clear definition of the design objectives. While in analysis and evaluation, the designer using these knowledge and simple diagrams, makes a first draft of the design in the form of diagrams. This has to be thoroughly analyzed to verify that it meets the design objectives and that it is adequate but not over designed. A systematic approach is useful only to the extent that the designer is presented with a strategy that he can use as a base for planning the required design strategy for the problem at hand. A design problem is not a theoretical construct. Design has an authentic purpose - the creation of an end result by taking definite action or the creation of something having physical reality. The design process is by necessity an iterative one. The software design should describe a system that meets the requirements. It should be thorough, in-depth, and complete and use standardized notations to depict the structure and systems. According to Ford (1991, p. 52):

The process of design consists of taking a specification (a description of what is required) and converting it into a design (a description of what is to be built). A good design is complete (it will build everything required), economical (it will not build what is not required), constructive (it says how to build the product), uniform (it uses the same building techniques throughout), and testable (it can be shown to work).

Most software engineers agreed that software design integrates and utilizes a great deal of the basics of computer science and information systems; unless one can apply the knowledge gained to the design of quality software, one cannot truly be considered to have mastered software design. Design processes are typically part of the larger SAND methodology.

THE ROLE OF SAND METHODOLOGY

With the advance of technology in the twentieth century and increasing reliance on IS or software, it is important that sound engineering practice is applied to the design and production of IS. Every intellectual discipline is characterized by fundamental concepts and specific techniques. Techniques are the application of the fundamental concepts. Though techniques can evolve or change, the underlying fundamental principles remain the same.

Why is there a need for a SAND methodology? A SAND methodology can provide a checklist of actions, a logical configuration of activities and may provide for the use of particular forms of notation or diagrams. In addition, it provides a common standard notation for communication between participants. In addition, the use of a "standard" methodology makes it easier for maintenance designers to understand the ideas and models used by the original designer; it helps them to model and assess the likely effects of any changes they might need to make. The use of a methodology also assists in the handover of a design since it would aid in the reconstruction of the necessary models and the understanding of the reasons behind particular choices. It should be noted that design methodologies are also apt to be strongly oriented toward one domain of application, through the criteria they use and the weighting these criteria are given.

The role of a SAND methodology is best understood in the context of the other activities involved in the production or development of a software system. The SDLC provides a standard operating procedure and framework that support a structured, engineered approach to system development. A recommended distribution of effort across the SDLC phases is called the 40-20-40 rule (Burch, 1992), which emphasizes strong front-end design and back-end testing whereas the mechanical programming task is de-emphasized. If the design is error-free, programming and testing should follow with little difficulty. In addition, the more critical and larger a program is, the more effort will be devoted to designing and testing phases.

Given that design plays an important role in SAND, it is necessary therefore, to try to improve it through appropriate research. Moreover, to do this, we must have a good scientific understanding of SAND on which to provide systems analyst/programmers a better heuristics to determine the efficacy of each of these methodologies and enable higher analyst/programmer efficiency and effectiveness in conducting SAND. This research project is a small step in this direction.

There are two broad categories of design methodologies: the systematic and the formal types. As the name imply, the formal type makes extensive use of mathematical notations for the object transformations and for checking consistencies. The systematic type is less mathematical and is consist of the procedural component, which prescribes what action or task to perform and the representation component, which prescribes how the software structure should be represented. One of the major methodologies for systems design today is the object-oriented system analysis and design (OOSAND). The next section will discuss specific principles of the processes in the methodology.

Object Oriented Analysis and Design

The explosive growth of the object-oriented technology in the 1990s has resulted in a paradigm shift in systems development methodologies. The Unified Process (Jacobson, 1999; Kruchten, 2000) is a direct result of the shift initiated by object-oriented technology. There are four main iterative phases in the Unified Process (UP) - Inception, Elaboration, Construction and Transition. Each of the phases has associated workflows and can have multiple iterations. See Figure 1.

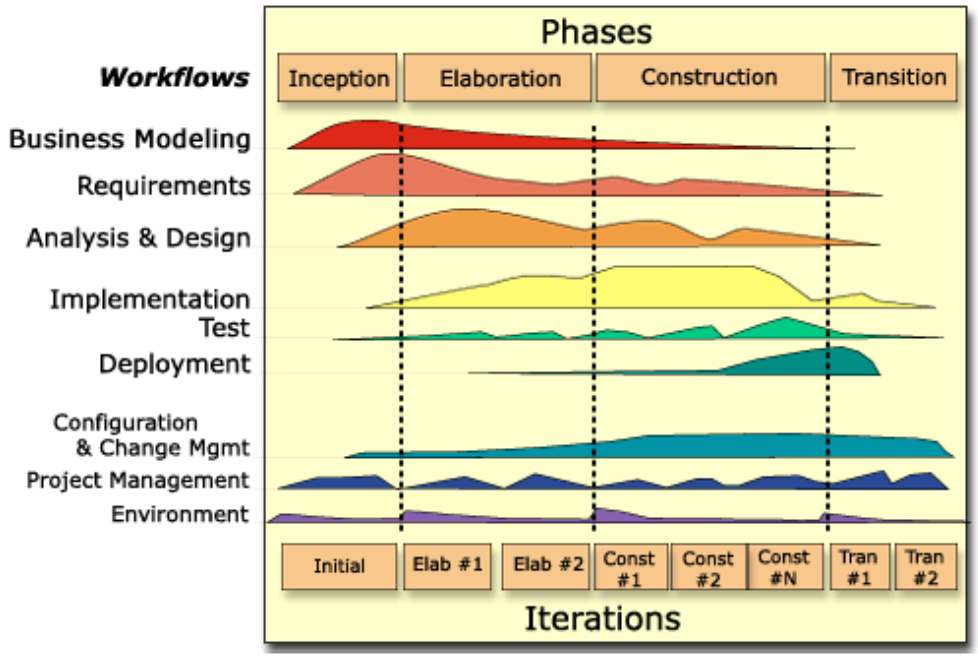

Figure 1. The Rational Unified Process (from Rational University)

The Inception phase includes tasks such as defining the scope, planning, feasibility studies and system investigation to model business system processes and define requirements. The OOSAND is an object-oriented process that takes place in the Elaboration phase. The Elaboration phase includes modeling the system processes as usecases, realizing the usecases and extracting a preliminary set of requirements from them. The usecases are determined from information gathered during the system investigation. Usecase descriptions (static notation) are defined and the sequence diagrams (dynamic notation) derived. The class diagrams are inferred from the sequence diagrams. The OOSAND is conducted for two main purposes: discover classes and to assign responsibilities between the classes. OOSAND uses the Unified Modeling Language (UML) (Rumbaugh, Jacobson & Booch, 2005) as diagrammatic representation for modeling the system. UML had been embedded in a number of system design tools. Rational Rose is a systems design tool that uses UML to construct the artifacts of OOSAND. The goal of the Elaboration phase is to produce a system design in the form of a class diagram that meets the requirements. In the Construction phase, programming code is written with the class diagram as blueprint and is tested based on requirements. When the software system has completed all the tests successfully, it is released and deployed on the clients' computers. The Transition phase involves releasing the software system to the client.

The OOSAND approach is based on modeling the problem in terms of its objects, the operations performed on them and the interfaces between objects to develop software. Its emphasis is on the object, which is an information item or a consumer or producer of information (Pressman, 1992). A class is a set of objects that share a set of common structure and behavior. A class represents a type and an object is an instance of a class. A class contains three items: class name, list of attributes, and list of operations. Thus, a class represents a set of objects with similar attributes, common behavior and common relationships to other objects. The derivation of subclasses from a class is called inheritance. A subclass may have a few super-classes, thus multiple inheritance. The ability of any objects to respond to the same message and of each object to implement it appropriately is called polymorphism. Relationships describe the dependencies between classes and other classes. The software design structure (class diagram) uses association mechanisms for representing the data structure, specifying the operations and the execution procedure.

Object Oriented Analysis (OOA) begins with an analysis of the system to establish its boundaries and the different uses of the system - its usecases. The usecases are modeled and documented as a usecase description of the procedural flows. The relevant nouns used in the uscase descriptions are candidate names for classes. Classes are then identified and sequence diagrams used to denote the operations that occur in the flows of each usecase. In Object Oriented Design (OOD), the classes with its data and operations are extracted from the sequence diagrams to form the class diagrams and finally, the class diagram is analyzed and structured by means of various association strategies to derive a system design. See Figure 2 for an illustration of the relationships between the UML artifacts in OOSAND.

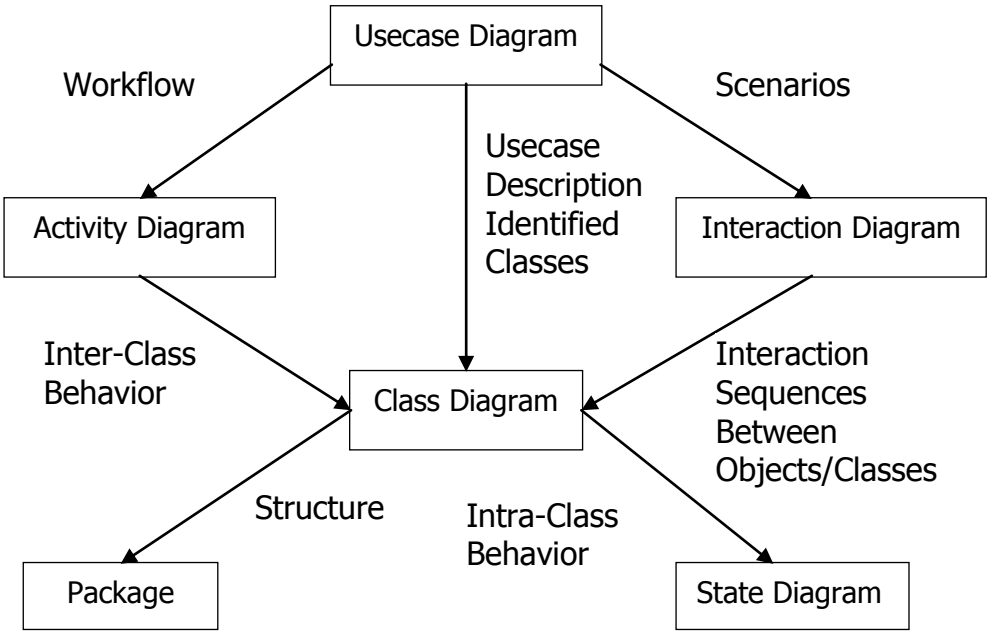

Figure 2. Relationship between UML Artifacts

OOD provides a mechanism that encompasses three important concepts in software design: modularity, abstraction, and encapsulation (also called information hiding). OOD concepts were first introduced by Abbot (1983) and were subsequently enhanced by Booch (1986, 1990). OOD is basically an approach that models the problem in terms of its objects and the operations performed on them. OOD decomposes the system into modules where each "... module in the system denotes an object or class of objects from the problem space" (Booch, 1986, p213). The modularity criteria from Structured Design (SD) (Peters, 1981) still apply in OOSAND.

Two categories of criteria are involved: the connections to other modules (coupling) and the intra-module unity (cohesion). The system level criterion, coupling provides a way of evaluating the inter-dependencies between modules. As modules are the building blocks of a software system, their relationships will determine how well the system can be maintained or changed. If the modules are highly interdependent on one another, it will be more difficult to make changes to one module without affecting the others. Conversely, if the modules are highly independent from one another, it will be easy to maintain and changes can be made on one module without affecting the others. The single module criterion cohesion provides a way of evaluating the functional connections between its processing elements. The most desirable cohesion is one where a module performs a single task with individual data elements. The least desirable cohesion, on the other hand, is one where a module performs a few different tasks with unrelated data structures.

A good system design will have strong cohesion and weak coupling. Two classification spectrums can be derived based on the coupling and cohesion characteristics. From Myers (1975) and, Yourdon and Constantine

(1978), a summary of modularity categories from good to bad, is given below:

Coupling Categories

1. Data: all communications between modules is through data element arguments.
2. Stamp: communication includes a data structure in the arguments (some fields are not needed).
3. Control: an argument from a module can control the flow of the other, for example, a flag.
4. External: they reference an externally declared data element.
5. Common: they reference an externally declared data structure.
6. Content: one references the contents of the other.

Cohesion Categories

1. Functional: they perform a single specific function.
2. Clustered: it is a group of functions sharing a data structure usually to hide its representation from the rest of the system; only one function is executed at each call, for example, the symbol table with insert and look-up functions.
3. Sequential: it consists of several functions that pass data along, for example, update and write a record.
4. Communicational: it consists of several logical functions operating on the same data, for example, print a file.
5. Procedural: its elements are grouped in an algorithm, for example, body of a loop.
6. Temporal: its functions related in time, for example, initialization.
7. Logical: it can perform a general function where a parameter value determines the specific function, for example, general-error-routine called with an error code.
8. Coincidental: no relationship between module elements that are grouped for packaging purposes.

The intention is to measure the program modules in terms of its cohesion and coupling. Its objective is for each module to perform a single specific function and that all its argument are individual data elements. Using these criteria, some design "quality" might be sacrificed as the result of the design trade-off analyses.

OOD is based on the concepts of: objects and attributes, classes and members, wholes and parts. Objects are basically a producer or consumers of information or an information item. Objects represent concrete entities, which are instances of one or more classes. All objects encapsulate data (the attribute values that define the data), other objects (composite objects can be defined), constants (set values), and other related information. Objects not only encapsulate data attributes, which can be data structures or just attributes, but also operations, which are procedures. The object consists of a private data structure and related operations that may transform the data structure. Operations contain procedural and control constructs that may be invoked by a message, that is, a request to the object to perform one of its operations. Encapsulation means that all of this information is packaged into a single name and can be re-used. Object oriented design encompasses data design, architectural design and procedural design. By identifying classes and objects, data abstractions are created; by coupling operations to data, modules are specified and a structure for the software is established; by developing a mechanism for using objects (for example, passing of messages) interfaces are described. The object that receives a message will then determine how the requested operation is to be performed. By this means, information hiding (that is, the details of implementation are hidden form all the elements outside the object) is achieved. Also objects and their operations are inherently modular, that is, software elements (data and process) are grouped together with a well-defined interface mechanism. As a result, a principal benefit of object-oriented software is that it is readily reusable.

Thus, the identification of classes and assignment of responsibilities are primary objectives of OOA and acts as a springboard for OOD. OOD creates a model of the ideal world and maps it to the software structure. Even though OOD provides the mechanism to partition the data and its operations to derive the system design, its representations are prone to have programming language dependency.

DISCUSSION

Due to the fact that methodologies have been developed from different milieu specifically to address certain problems or groups of problems, there is no common baseline on which to evaluate a methodology. However, the methodologies can be analyzed based on some characteristics. A tabulation of the characteristics of OOSAND is shown in Table 1.

Table 1. Characteristics of OOSAND

Characteristics	OOSAND
SDLC	Unified Process
Principal Artifacts	Usecase, Usecase Description, Sequence Diagram, Class Diagram
Driver	Objects
Data Structure	Class Diagram
Processes	Usecase Modeling, Workflow Modeling
Program Structure	Abstraction, Hierarchy, Encapsulation, Modularity
Modularity Structure	Class Associations
Notation	UML
Programming Language	C++, Java, ADA
Advantages	Reusability
Disadvantages	Conceptualizing Objects

A list of the principal processes in OOSAND is illustrated in Table 2.

Table 2. OOSAND Processes in the System Development Life Cycle (SDLC) Model.

Unified Process	**OOSAND**
	Principal Processes
Inception	Business Modeling, Systems Investigation, Requirement Definition
Elaboration	Usecase Modeling, Workflow Modeling (Usecase Description), Scenario Modeling (Sequence Diagram), Class Modeling (Class Diagram), Classes Association
Construction	Coding, Testing
Transition	Deployment

The underlying principles of OOSAND was analyzed and examined for a better understanding of the basis for the methodology. However, if another methodology is a viable option, it is possible at the initial phases of a project to consider a first iteration design using the alternate methodology (for example, Structured Analysis and Design or the Agile methodology). Generally, alternative design allows for important trade-off analysis before coding the software. Thus, familiarity with several methodologies makes creating competitive designs more logical and systematic with less reliance on inspiration.

CONCLUSION

This paper has critically examined and analyzed the OOSAND methodology so as to expose the underlying principles and rationales of their inherent processes. This paper focused on the processes of the methodology and has examined and highlighted the pragmatic issues of the methodology. With a better understanding of the methodology, its domain of application can be more effectively applied or more accurately defined. The significance of this research is that the knowledge gained in this exercise will provide structured systems analyst/programmers a better heuristics to migrate legacy systems developed by Structured Analysis and Design to the new object-oriented systems that uses OOSAND and enable higher analyst/programmer efficiency and effectiveness in conducting SAND.

REFERENCES

1. Abbot, R.J. (1983). Program Design by Improved English Descriptions. *Communications of the ACM*, 26(November), 882-894.
2. Boehm, B. W. (1988) A Spiral Model for Software Development and Enhancement. *IEEE Computer*, 21(May), 1988, pp. 61-72.
3. Booch, G. (1986). Object Oriented Development. *IEEE Transactions on Software Engineering*, SE-12(February), 211-221.
4. Booch, G. (1990). Object Oriented Design, Benjamin-Cummings, Redwood City, CA.
5. Booch, G, Rumbaugh, J., Jacobson, I. (1999) *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.
6. Budgen, David. (1989). Introduction to Software Design, Technical Report SEI-CM-002, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA., p 6.
7. Coad, P., & Yourdon, E. (1991). *Object-Oriented Analysis*. (2nd ed.). Yourdon Press/Prentice Hall, New York.
8. DeMarco, T. (1979). *Structured Analysis and System Specifications*, Yourdon Press, Englewood Cliffs, N.J.
9. Ford, Gary. (1991). 1991 SEI Report on Graduate Software Engineering Education, Technical Report SEI-91-TR-2, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, 52.
10. Jacobson, I., Booch, G., and Rumbaugh, J. (1999) *The Unified Software Development Process*. Reading, MA: Addison-Wesley.
11. Kruchten, P. (2000) *The Rational Unified Process - An Introduction*. 2nd edition. Reading, MA: Addison-Wesley.
12. Myers, G.J. (1975) *Reliable Software Through Composite Design*, Petrocelli/Charter, New York.
13. Page-Jones, M. (1980a). *Transform Analysis. The Practical Guide to Structured Systems Design*, Yourdon Press, Englewood Cliffs, N.J., 181-203.
14. Page-Jones, M. (1980b). *Transaction Analysis. The Practical Guide to Structured Systems Design*, Yourdon Press, Englewood Cliffs, N.J., 207-219.
15. Peters, L.J. (1981). *Software Design: Methods and Techniques*, Yourdon Press, New York.
16. Pressman, R.S. (1992) *Software Engineering: A practitioner's Approach*, McGraw-Hill, pp. 34 - 36.
17. Royce, W.W. (1970). Managing the Development of Large Software Systems: Concepts and Techniques. Proceedings WESCON, pp 1-9.
18. Rumbaugh, J., Jacobson, I., Booch, G. (2005) *The Unified Modeling Language Reference Manual*, Second Edition. Reading, MA: Addison-Wesley.
19. Stevens, W., Myers, G., & Constantine, L. (1974). Structured Design. *IBM Systems Journal*, International Business Machines Corporation, 13(May), 115-139.
20. Svoboda, Cyril P. (1990) Structured Analysis: System and Software Requirements Engineering, IEEE Computer Society Press, Los Alamitos, California.
21. Wirfs-Brock, R.J., & Johnson, R.E. (1990). Surveying Current Research in Object Oriented Design. *Communications of the ACM*. 33(September), 104-124.
22. Yourdon, E., & Constantine, L. (1978). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Second Edition, Yourdon Press, New York.

